Proceedings of the 2023 Winter Simulation Conference
C. G. Corlu, S. R. Hunter, H. Lam, B. S. Onggo, J. Shortle, and B. Biller, eds.

EFFICIENT HYBRID SIMULATION OPTIMIZATION VIA GRAPH NEURAL NETWORK METAMODELING

Wang Cen
Peter J. Haas

Manning College of Information and Computer Sciences
University of Massachusetts Amherst
140 Governors Drive
Amherst, MA 01003, USA

ABSTRACT

Simulation metamodeling is essential for speeding up optimization via simulation to support rapid decision making. During optimization, the metamodel, rather than expensive simulation, is used to compute objective values. We recently developed graphical neural metamodels (GMMs) that use graph neural networks to allow the graphical structure of a simulation model to be treated as a metamodel input parameter that can be varied along with scalar inputs. In this paper we provide novel methods for using GMMs to solve hybrid optimization problems where both real-valued input parameters and graphical structure are jointly optimized. The key ideas are to modify Monte Carlo tree search to incorporate both discrete and continuous optimization and to leverage the automatic differentiation infrastructure used for neural network training to quickly compute gradients of the objective function during stochastic gradient descent. Experiments on stochastic activity network and warehouse models demonstrate the potential of our method.

1 INTRODUCTION

Stochastic simulation and optimization can be combined to support decision making under uncertainty for complex systems. Optimization via simulation (OvS) has been an essential tool in manufacturing, logistics, healthcare, transportation, and many other fields. When the simulation model is complex and computationally expensive, however, using it directly in the optimization process can be highly time-consuming, rendering naive OvS unsuitable in tactical or operational settings where decisions must be made quickly. In this situation, simulation metamodels can be highly desirable.

Simulation metamodeling A simulation metamodel captures the statistical relationships between simulation inputs and outputs (Barton 2020); the output is usually a real-valued performance measure of interest, such as expected daily cost or expected average customer delay. A metamodel is developed by running a large number of simulations of the system under various conditions and then using the data collected from these simulations to create a statistical model. Once the simulation metamodel is developed, optimization algorithms are used to determine the combination of input values that yields the best result using the metamodel in place of the original simulation model; this speeds up optimization because metamodels execute orders of magnitude faster (Barton 2020; do Amaral et al. 2022). Traditional choices include polynomial regression metamodels, Gaussian process metamodels, and neural networks; the latter can learn a highly nonlinear input-output mapping from the simulation data.

Limitations of traditional metamodeling methods Standard metamodeling methodology has a couple of key limitations. First, an input x must be a fixed-length vector of real or integer-ordered values. Thus a metamodel cannot easily represent variations in system *structure*, such as changes to the layout of aisles, bins, or items in a warehouse, or changes to the sequencing and synchronization constraints on a

set of activities that comprise a task such as manufacturing an item. In principle, structural aspects can be encoded via discrete variables—e.g., an adjacency matrix can describe bin layouts. However, such naive representations are highly inefficient, do not easily allow for structures whose representations have varying dimensions, and are further hampered by a lack of "permutation invariance", with mere relabeling of nodes non-intuitively yielding differing predictions (Marti 2019).

Graphical metamodels To deal with the foregoing limitations, Cen and Haas (2022) proposed the use of *graph neural networks* (GrNNs) to represent the graph structure of a simulation model in an effective manner. This approach leverages the fact that many simulations of real-world systems have aspects that can be represented by a graph structure. For example, Haas and Shedler (1991) show that a broad range of discrete-event systems can be represented via a particular class of graphs: stochastic Petri nets. Incorporating this structural information yields insights into the inner workings of a system and leads to more accurate simulation metamodels. The resulting *graphical metamodels* (GMMs) can be further enhanced with generative neural network components, allowing for metamodeling of entire probability distributions as well as stochastic processes.

Hybrid optimization The inputs to a GMM typically include real-valued parameters—such as service rates or travel times—along with the graph structure, which is discrete. More formally, Let $\mathscr{M}$ be a discrete set of simulation models, each with its own distinct graph structure, and let $\Gamma_M \subseteq \Re^j$ be the set of feasible values for the $j = j(M)$ scalar-valued simulation input parameters required when simulating model $M \in \mathscr{M}$. Let $C(\gamma, M)$ be a cost function that depends on both the simulation model M and its vector of input parameters γ. Then the hybrid optimization problem generally has the form: $\min_{M \in \mathscr{M}, \gamma \in \Gamma_M} C(\gamma, M)$. Note that we require the capability to systematically generate elements of $\mathscr{M}$ when learning the metamodel.

This setup begs the question of whether efficient joint optimization over both types of inputs is possible. These types of hybrid discrete-continuous optimization problems have received relatively little attention in the literature. They are extremely challenging because the discrete part of the search space can be exponentially large, and each discrete element requires optimization of its accompanying continuous parameters, exacerbating the computational burden.

In this paper, we present a novel Hybrid OvS (HOvS) algorithm for joint optimization of discrete and continuous variables, based on a graphical metamodel. The key idea is to modify *Monte Carlo Tree Search (MCTS)*—an adaptive optimization algorithm designed to handle arbitrary discrete data—to incorporate continuous optimization as well. A Monte Carlo tree search initially performs a random search of the decision space but, as the search process continues, focuses on promising regions (Fu 2018). In our novel *hybrid MCTS* (H-MCTS) algorithm, the values of the discrete decision variables are first determined in a branching search tree as in the standard algorithm, but at each terminal leaf—which corresponds to a given choice for all of the discrete variables—the continuous decision variables are optimized using gradient descent. The optimization result at a leaf serves as a reward signal that guides the optimization process going forward; a good result means that the algorithm will focus more on that region of the tree. If statistical guarantees on the final result are desired, the selected system can be simulated; moreover, ranking and selection techniques can be applied as in Boesel et al. (2003).

We are aware of only one other hybrid optimization algorithm for neural network metamodels, due to Wang (2005). The algorithm is designed for integer and real-valued parameters: it first converts all integer values to real numbers and then uses a continuous genetic algorithm for optimization, with the "crossover" operation (creating offspring from a couple of parents) implemented as a weighted combination of the parents. The algorithm cannot deal with discrete graph structures, however, since weighted combinations of graphs, especially with different numbers of nodes and edges, are not well defined.

Gradient estimation We use gradient descent to optimize the continuous decision variables at a leaf during our Monte Carlo tree search. Gradient estimation is nontrivial in our GMM metamodel setting: methods such as IPA or likelihood-ratio estimation are not applicable because detailed information about the simulation model is not available. One naive approach would use finite difference methods to estimate the gradient, with all the computational effort that this technique entails; see Section 6.1. Another possibility

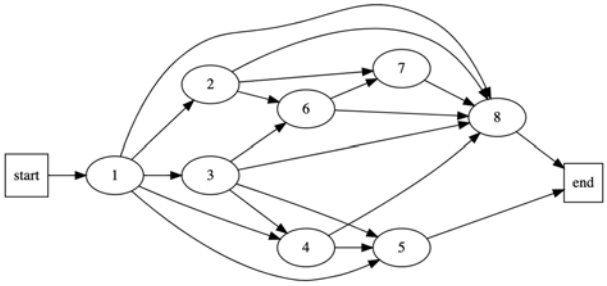


Figure 1: Stochastic activity network for a manufacturing task.

would be to build an additional metamodel to estimate the gradient of the objective function, given values of the input parameters and the graph structure. Constructing such a model would typically be very expensive because the gradient would need to be estimated at every training point.

We propose to leverage the *Automatic Differentiation (AD)* functionality provided by modern neural network frameworks to perform efficient and inexpensive gradient computation. Specifically, modern neural network frameworks such as PyTorch and TensorFlow employ highly optimized AD algorithms to compute the gradient of the loss function with respect to the network parameters (weights and biases) during the process of network training via gradient descent. The key idea is that once the neural network is trained, we can repurpose the AD infrastructure to easily and efficiently compute the gradient of the objective function in our continuous optimization problem with respect to the continuous decision variables. This general technique was originally developed in the neural-network setting by the explainable-AI community, where the task is to understand the importance, with respect to a given image-classification result, of each pixel in the image (Simonyan et al. 2013). There has also been work on applying AD directly to various subclasses of simulation models; see, e.g., Ford et al. (2022), where it is sometimes called the "adjoint" technique. To our knowledge, this is the first application of the repurposing idea to neural-network-based metamodels in the OvS setting.

Paper Organization We first introduce a running example of a hybrid optimization problem for a stochastic activity network (SAN) in Section 2, and then give brief overviews of GMMs and AD in Sections 3 and 4. We next describe our modified Monte Carlo tree search algorithm in Section 5. Section 6 provides empirical results for the SAN hybrid optimization problem as well as a purely continuous and a purely discrete OvS problem. Of the latter two problems, the first is a special case of our SAN example, and the second involves layout design and item placement in a warehouse. For the purely continuous problem, we also compare AD gradient estimates to finite-difference estimates in terms of cost and accuracy. Experiments show that our optimization technique performs well, finding high-quality solutions to the novel hybrid optimization problem, while also performing satisfactorily on purely continuous and purely discrete optimization problems. We conclude in Section 7.

2 HYBRID OPTIMIZATION EXAMPLE: STOCHASTIC ACTIVITY NETWORK

We start by describing a hybrid optimization problem that will serve as a running example. Imagine that we are manufacturing custom items. The process for each item consists of a set of partially-ordered, synchronized activities that can be represented using a stochastic activity network (SAN) of the type described in Kim et al. (2007); see Figure 1. In this directed acyclic graph, each activity represents a step in the manufacturing process. The nodes represent activities and the directed edges indicate the order in which activities must be performed. Specifically, an activity cannot start until all its parent activities have completed. Moreover, each activity i has a random duration modeled by an exponential distribution with parameter λ_i. The value of the work rate λ_i depends on, e.g., the speed at which a robot is operated for activity i. Higher work rates (larger values of λ_i) correspond to faster activity completion but also to higher

costs (e.g., more electricity consumed or higher probability of defects). We want to adjust the work rate λ_i for each activity to minimize the total cost of completing the order-assembly tasks.

In addition to deciding the optimal choice of work rates, we can also decide to remove edges between activities. Removing an edge from an activity i to another activity j corresponds to using an external vendor to stockpile parts used in activity j that were formerly manufactured in-house via activity i. This type of action removes the dependency between the activities and may enable each activity to start sooner, potentially reducing the overall completion time. However, we incur an additional cost for using the vendor. So in this hybrid optimization problem, we must jointly optimize the work rates and the set of edges removed.

Consider a SAN with n activities and m possible edges between them, e.g., we have $n = 8$ and $m = 17$ in Figure 1. Let $\lambda = (\lambda_1, \ldots, \lambda_n)$ be the vector of work rates and $x = (x_1, \ldots, x_m)$ be a vector with $x_j = 1$ if we retain edge j and $x_j = 0$ otherwise. We can model the SAN as a Markov process $\{Z(t)\}_{t \geq 0}$, where $Z_i(t) \in \{0,1,2\}$ for $i \in [1..n]$, corresponding to the cases where activity i has not yet started (0), is in progress (1), or has completed (2). Then the task completion time $Y = Y(\lambda, x)$ is the hitting time of the absorbing state $s = (2,2,\ldots,2)$. Suppose we want to solve the problem: $\min_{x \in \mathscr{X}, \lambda \in (0,\infty)^n} C(\lambda, x)$, where

$$C(\lambda, x) = E[Y(\lambda, x)] + \alpha \sum_{j=1}^{n} (1 - x_j) + \beta \sum_{i=1}^{m} \lambda_i, \tag{1}$$

α, β are positive constants and $\mathscr{X} \subset \{0,1\}^m$ is a constraint set which ensures that, after removing edges, every activity node has at least one outgoing edge. Note that, with respect to our general definition of the hybrid optimization problem, there is a 1-to-1 correspondence between a feasible vector $x \in \mathscr{X}$ and a model $M \in \mathscr{M}$; moreover, the constraint set Γ_M for the continuous parameters is the same for each M.

We can train a GMM metamodel to predict the completion time of a SAN like the one shown in Figure 1. Our training dataset comprises randomly generated SANs, each with its expected completion time estimated over a small number of simulation replications; see Cen and Haas (2022) for details.

Crucially, we can train a *single* GMM to predict completion times for a broad range of SAN graphs having different numbers of nodes and edges. Thus we can not only solve the above optimization problem for the SAN in Figure 1, but can use the same GMM to optimize previously-unseen SANs as they are encountered, as expected when manufacturing custom items. One proviso is that a new SAN cannot be wildly different from the SANs used for training, and so the flexibility of the metamodel depends on the breadth of the training data. We again note that traditional metamodeling cannot handle varying SAN graphs in this way. Even if the number of nodes and possible edges are fixed, using a traditional metamodel would be suboptimal because (1) it is computationally cumbersome and (2) it is less accurate because the structural dependencies in the SAN, e.g., critical paths, would not be explicitly represented.

3 GRAPH NEURAL NETWORK METAMODELS

We next provide a brief introduction to graphical metamodels (GMM); see Cen and Haas (2022) for details. A GMM works by first transforming a discrete graph structure into a high-dimensional embedding, such that graphs having similar structures are close in the high-dimensional space. Next, the embedding is fed into a Multilayer Perceptron (MLP) neural network to predict a performance measure we are interested in.

Specifically, to use a GMM as a simulation metamodel, we extract the relevant graph structure from the original simulation model, such as a task graph or warehouse layout. Each node in the graph is then annotated with a vector of numerical input values or "features". For instance, in the case of a SAN, each node is annotated with a work rate λ_i. This annotated graph is then encoded into a high-dimensional vector or “embedding” by a Graph Neural Network (GrNN) through a process called message passing, which summarizes the information in the graph. The GrNN implicitly captures the “important” aspects that differentiate one annotated graph structure from another—including the work rates for the activities—in an effective manner.

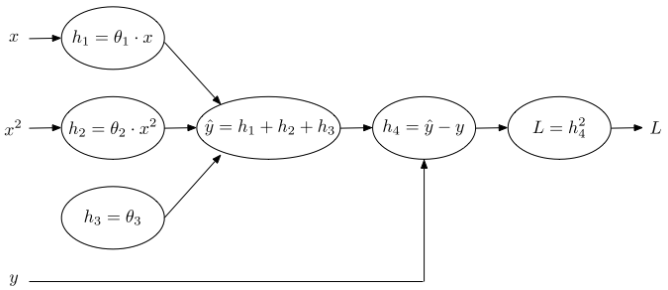


Figure 2: Stylized neural network for computing loss L.

The encoded graph representation h_G is then used as input to a MLP that predicts the output y of interest, such as an expected task completion time $y = E[Y(\lambda, x)]$. Thus GMMs provide a simulation metamodeling framework that can leverage structural information to capture a broad variety of structurally varying simulation models in a single metamodel. GMMs can be efficiently trained by gradient descent, as discussed in the next section.

4 AUTOMATIC DIFFERENTIATION

Automatic differentiation (AD) is a method used for efficiently computing the derivative of a complex mathematical function with respect to its input variables. In neural networks, it is used to compute the gradient of the loss function during the training phase, where the network parameters θ are optimized via gradient descent to minimize the loss over the training data. One of its key advantages is that the user only needs to specify the *forward pass* that determines how the output is computed; this is embodied in the structure of network. The *backward pass*—i.e. the computation of the gradient of the output with regard to the inputs—is automatically derived and computed via a standard neural network library.

AD exploits the fact that a neural network breaks down the computation of a complex function into a sequence of small computations performed by the neurons; each neuron applies a function corresponding to an "elementary operation". Such a function typically involves matrix multiplications and vector additions, and then application of a nonlinear "activation function" such as a ReLU or sigmoid function. Crucially, the derivative of each elementary function can be computed analytically, and routines for evaluating the analytical expressions are bundled with common deep learning frameworks such as PyTorch, TensorFlow and Jax. Then, to automatically compute the derivatives of a complex function with regard to the individual parameters, we use the chain rule of differentiation.

We illustrate the AD technique using the stylized neural network given in Figure 2. The elementary operations are simply addition, multiplication, and exponentiation. In our toy example, the goal is to optimize the neural net parameters $\theta = (\theta_1, \theta_2, \theta_3)$ to fit the function $y = \theta_1 x + \theta_2 x^2 + \theta_3$ to a training dataset comprising tuples of the form (x, x^2, y). Given a training point (x, x^2, y) and current parameters $\tilde{\theta}$, the forward pass computes the loss $L = (\hat{y} - y)^2 = (\tilde{\theta}_1 x + \tilde{\theta}_2 x^2 + \tilde{\theta}_3 - y)^2$. Then, using the chain rule, we can compute the gradient of the loss function with respect to, e.g., the parameter θ_2 at point x automatically in a backward pass:

$$\frac{\partial L}{\partial \theta_2} = \frac{\partial L}{\partial h_4} \frac{\partial h_4}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h_2} \frac{\partial h_2}{\partial \theta_2} = 2 h_4 x^2,$$

where h_4 has been computed during the forward pass and cached. The parameter $\tilde{\theta}_2$ is then updated as $\tilde{\theta}_2 \leftarrow \tilde{\theta}_2 - a \partial L / \partial \theta_2$, where a is a step size; the other components of $\tilde{\theta}$ are updated similarly. Our implementations employ the Adam algorithm (Kingma and Ba 2014), with default hyperameters, to choose the step size. During the training of a neural network, this process is iterated many times, converging to optimal parameter values θ_1^*, θ_2^*, and θ_3^* with a corresponding fitted model $y = \theta_1^* x + \theta_2^* x^2 + \theta_3^*$.

Our key observation is that we can re-purpose the same machinery to compute gradients during simulation optimization. Suppose, for example, that after training the foregoing network, we want to compute the

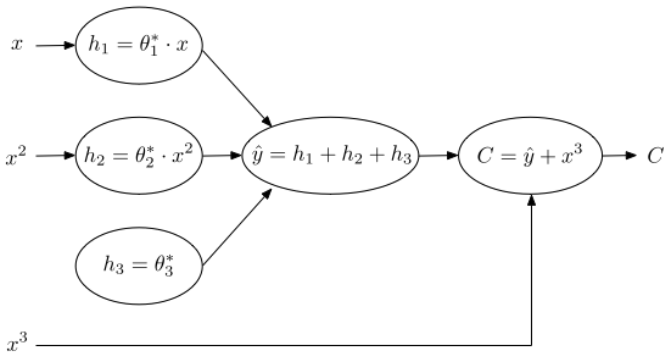


Figure 3: Stylized network for computing $\partial C/\partial x$.

derivative of the cost function $C = y + x^3$ with respect to x. We can then run a forward and backward pass on the modified network given in Figure 3 to compute both the cost C and its derivative. In the backward pass, we find that

$$\frac{\partial C}{\partial x} = \frac{\partial \hat{y}}{\partial x} + \frac{\partial x^3}{\partial x} = \frac{\partial \hat{y}}{\partial h_1}\frac{\partial h_1}{\partial x} + \frac{\partial \hat{y}}{\partial h_2}\frac{\partial h_2}{\partial x} + \frac{\partial x^3}{\partial x} = \theta_1^* + 2\theta_2^* x + 3x^2.$$

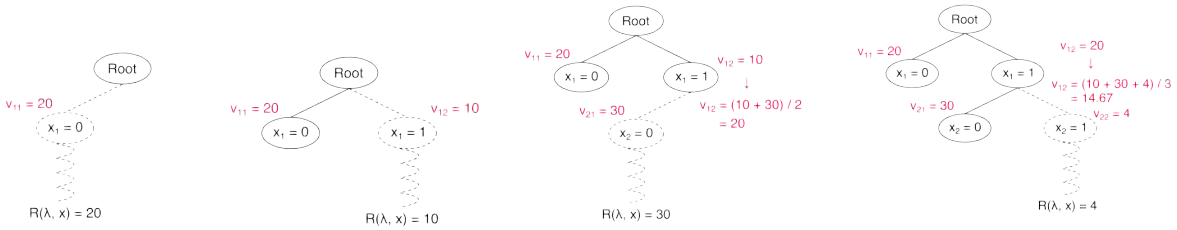
We can now use gradient descent to solve continuous simulation optimization problems that arise within the hybrid Monte Carlo tree search that is described in the next section.

5 HYBRID MONTE CARLO TREE SEARCH

Monte Carlo Tree Search (MCTS) is a decision-making algorithm commonly used in artificial intelligence and game theory. It is a heuristic search algorithm that uses statistical sampling and simulation to explore large decision trees, commonly used in games that have a large number of possible actions, such as chess, Go, and poker. The key idea of MCTS is to iteratively build a search tree by simulating a large number of random games and then using the results to guide the search towards the most promising moves or solutions. In the case of games, the results usually refer to the win (+1) or loss (-1) of the game. In this paper, we employ MCTS to determine the values of the discrete variables in an optimization problem. To adapt MCTS for hybrid optimization problems, we make the key change that when MCTS reaches the terminal leaf, i.e., when values for all discrete variables have been selected for a proposed solution, gradient descent as described in Section 4 is performed to optimize the continuous variables and the final objective value is back-propagated in place of a game result. Here we give a brief description of H-MCTS; see Fu (2018) for more details on standard MCTS. The standard MCTS algorithm is described in terms of maximizing a reward function R, and we will adopt this convention here. When we apply MCTS to cost minimization problems in subsequent sections, we will simply maximize the negative cost, i.e., $R(\lambda, x) = -C(\lambda, x)$.

H-MCTS follows an iterative four-step process that builds an annotated search tree by selecting, expanding, optimizing, and back-propagating the results of the optimization. Each node i is annotated by its current estimated objective value v_i, computed as the average of all objective values over observed paths that pass through node i. Initially there is a single root node (level 0 of the tree), and the tree grows as existing nodes are expanded by adding child nodes. During each iteration, the algorithm selects the next node to expand by balancing exploration and exploitation based on the current state of the tree. This process continues until a predetermined stopping criterion is met, e.g., a specified number of steps have been executed or a specified time limit has been reached. In more detail, the steps are as follows.

Selection MCTS starts each iteration by selecting a node from the existing search tree. Starting from the root node, the algorithm follows a path down the tree based on a selection policy. This policy balances between exploration of new nodes and exploitation of previously visited nodes with high values.



(a) Iteration 1. (b) Iteration 2. (c) Iteration 3. (d) Iteration 4.

Figure 4: State of the search tree at the end of the ith MCST iteration for $i = 1,2,3,4$.

The selection process continues until a leaf node is reached, i.e., a node that has not been fully expanded. We use a recent version of MCTS due to Danihelka et al. (2022) that maintains an averaged objective value v_i for each search node i. This variant selects a child j by computing $\text{argmax}_j(v_j + \varepsilon_j)$, where each ε_j is an independent sample from the standard Gumbel distribution with pdf $f(x) = \exp(-x - e^{-x})$.

Expansion Once a leaf node is reached, the next step is to expand the node by adding a child node. This is done by applying a valid action to the current state represented by the leaf node, creating a new child node that represents the resulting state. A given root-to-leaf path through the tree thus corresponds to a sequence of actions. In our setting, an action on a node at level i of the tree corresponds to selecting a feasible value for the ith discrete variable.

Optimization After a new child node is added at level i, the algorithm computes an objective value v for the node as follows. First, a complete feasible solution $\bar{x} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)$ is generated. The values of $\bar{x}_1, \ldots, \bar{x}_i$ are determined by the path from the root to the child node, and the values of $\bar{x}_{i+1}, \ldots, \bar{x}_n$ are selected randomly and independently. This process can be viewed as taking a random walk down the tree from level i to terminal level n. Then we use gradient descent on the GMM to maximize the reward function $R(\lambda, \bar{x})$ with respect to λ and then set v equal to the optimal objective value $R(\lambda^*, \bar{x})$, which will be used to update the nodes along the path from the root to the new child node.

Backpropagation The final step of H-MCTS is to propagate the reward information v back up the tree, updating the statistics for each of the nodes that were visited along the path from the root to the new child node. The statistics are updated recursively from the new child node up to the root. The value for the new child node at level i is simply the objective value v as described above. For a node at level $j < i$ on the path from the root to the new child node, the node value is updated to be the cumulative average of its original value plus all values propagated from its children so far.

The MCTS process continues until a specified number of iterations have occurred or a given amount of time (determined by the computation budget) has elapsed.

Figure 4 illustrates the first four iterations of H-MCTS for a hypothetical optimization problem where we want to maximize a reward function $R(\lambda, x)$; here λ is a vector of continuous parameters and $x = (x_1, x_2, \ldots, x_n)$ is a vector of n binary variables. We denote by v_{ij} the value for the jth node created at level i of the tree. At the start of iteration 1, the root node has yet been expanded. We randomly add the child node $x_1 = 0$, then set $\bar{x}_1 = 0$, randomly choose values for $\bar{x}_2, \ldots, \bar{x}_n$, and perform a gradient descent on the λ variables to obtain a reward of 20, which is then back-propagated to node $x_1 = 0$. In iteration 2, we add the remaining child node $x_1 = 1$, set $\bar{x}_1 = 1$, randomly set values for $\bar{x}_2, \ldots, \bar{x}_n$, solve the resulting continuous optimization problem, and obtain a reward of 10, which is then back-propagated to node $x_1 = 1$. In iteration 3, since the root node is fully expanded, we decide on the next node to expand by generating Gumbel samples ε_1 and ε_2, say 5 and 20, and then compute $v'_{11} = v_{11} + \varepsilon_1 = 20 + 5 = 25$ and $v'_{12} = v_{12} + \varepsilon_2 = 10 + 20 = 30$ and select the node with the maximum perturbed value, in this case node $x_1 = 1$, for expansion. This expansion results in creation of (randomly-selected) child node $x_2 = 0$. We then set $\bar{x}_1 = 1$, $\bar{x}_2 = 0$, randomly select

values for $\bar{x}_3$ through $\bar{x}_n$ and solve the continuous problem to obtain a reward value of 30, which is then propagated to node $x_2 = 0$; the propagation continues as node $x_1 = 1$ is updated by taking an average of 10 and 30. In iteration 4, suppose $x_1 = 1$ is again selected for expansion. Then $x_2 = 1$ is created. The ensuing reward of 4 is propagated to node $x_2 = 1$ and for $x_1 = 1$ the value is updated to $(10+30+4)/3$.

6 EXPERIMENTS

We now describe some preliminary experiments where we apply our H-MCTS approach to optimization over a GMM metamodel. To first verify that our approach can reasonably handle purely continuous and purely discrete optimization problems, we experiment with a continuous, simplified version of the SAN optimization problem given in Section 2 and a discrete optimization problem involving a warehouse layout. Finally, we apply our method to the full hybrid SAN optimization problem.

For all experiments, we used PyTorch version 1.13, with both the simulations and the H-MCTS algorithm implemented in Python. To train the neural networks, we used a consumer-grade NVIDIA GeForce GTX 1660 SUPER GPU. GrNNs were trained using $L = 4$ message-passing iterations. More details of training GMMs are available in Cen and Haas (2022). During optimization, the models are run on CPUs. The experiments were run on Amazon Web Services c6a.4xlarge instances.

6.1 Continuous Optimization: SAN Work Rates

We first trained a single GMM for all of our SAN optimization experiments in this section and in Section 6.3. We used 2,000 training points, each corresponding to a SAN where both structure and work rates were randomly generated; see Cen and Haas (2022) for details.

For our first experiment, we consider a special case of the SAN optimization problem given in Section 2 where the structure of the SAN is fixed as in Figure 1 and we are optimizing only the work-rate vector λ in order to minimize costs; i.e., we fix x at the value $\bar{x} = (1, 1, \ldots, 1)$, and set $\alpha = 0$ and $\beta = 6$ in the objective function (1). We restrict each λ_i to lie in the range $[0.25, 1.0]$ and take $\beta = 6$, so that our optimization problem is

$$\min_{\lambda} E[Y(\lambda, \bar{x})] + 6 \sum_{i=1}^{8} \lambda_i$$
$$\text{s.t. } \lambda_i \in [0.25, 1.0] \text{ for } i \in [1..8],$$

where, as before, $E[Y(\lambda, x)]$ is the expected completion time of the SAN under the work rates in λ and edge set indicated by x.

To verify that our method works as expected in the simple case of pure continuous optimization, we first examined an even simpler version of the SAN continuous-optimization problem where λ_3 through λ_8 are fixed at values that are randomly and independently sampled from a uniform distribution over $[0.25, 1.0]$ and then we optimize only λ_1 and λ_2. We used PyTorch to execute gradient descent with gradients computed via AD as in Section 4. For this two-parameter problem, we were able to use brute-force search over a high resolution 2D grid with roughly 450 grid points and 50,000 simulation replications at each grid point, to verify the solution. The optimization led to a solution with a cost of 20.19, with corresponding parameter values of $\lambda_1 = 0.3925$ and $\lambda_2 = 0.3233$. The brute force search using 50,000 replications yielded a solution with a cost of 20.11, corresponding to parameter values of $\lambda_1 = 0.4255$ and $\lambda_2 = 0.3774$. The difference between the two solutions was only -0.42%. Notably, the gradient descent approach was exceptionally fast, completing the optimization in roughly 0.01 seconds, while the brute force method had yet to complete a single simulation replication.

We then performed a full-scale experiment where we optimized all eight λ_i parameters. For verification of the result, the brute force approach was beyond reach, and we used random search over the metamodel as a simple verification baseline. The gradient descent returned a cost of 33.41 and the random search returned an objective of 33.60 with a budget of 1,000 SAN evaluations via the GMM. Our solution is

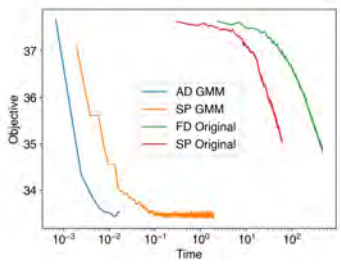


Figure 5: Objective value over time for the SAN continuous optimization problem using AD or SP gradient estimators on the GMM and FD or SP estimators on the original simulation model.

Table 1: The optimized λ values returned by gradient descent and random search.

	λ_1	λ_2	λ_3	λ_4	λ_5	λ_6	λ_7	λ_8
Gradient descent	0.4132	0.3236	0.3922	0.3436	0.3195	0.3623	0.3817	0.3774
Random	0.3597	0.2639	0.3165	0.3030	0.2597	0.3448	0.3663	0.3559

slightly better than the random search, by 0.56%. Again, the gradient descent finished within 0.02 seconds and only required roughly 10 SAN evaluations via the GMM. The values of λ_i are shown in Table 1.

In this same full-scale setting, we ran an experiment to compare three different methods of computing the gradient from a metamodel—AD, symmetric finite difference (FD), and simultaneous perturbation (SP) as in Spall (2003). The FD and SP difference-length parameter δ was 0.001, but our results were similar for $\delta \in [0.0001, 0.1]$. Based on 100 runs for each method, we found that AD, FD and SP on average took 2.11, 13.4 and 1.92 ms to compute the gradient. FD was the slowest method due to its requirement to evaluate the metamodel at $2 \times 8 = 16$ design points per gradient estimate. SP outperformed AD since AD incurs the overhead of computing the backward pass. On the other hand, the numerical values of the gradient obtained from AD and FD methods were very similar to each other but different from SP.

The fast speed of SP prompted the question of whether SP is also suitable for performing continuous optimization using a GMM. In a follow-on experiment, we used the foregoing gradient estimates inside a standard gradient-descent algorithm with a fixed step size of 0.001. Figure 5 indicates that AD converged to an optimal solution in approximately 0.01 seconds while SP converged in around 0.1 seconds, 10x slower. Although SP was slightly faster than AD at each iteration, the inferior quality of its gradient estimates caused slower overall convergence. For comparative purposes, the experiment also considered using FD and SP gradient estimators directly on the original SAN model, without employing the metamodel. Since executing the original SAN simulator is considerably slower than evaluating the metamodel, neither FD nor SP using the original model converged within the 100-second timeframe, illustrating the usefulness of performing optimization with GMMs when decisions are needed quickly.

6.2 Discrete Optimization: Warehouse Layout and Placement Optimization

Our next example is a discrete optimization problem involving a warehouse. The warehouse has 16 storage bins and 16 items; each bin holds one item. We denote by $I = \{1, 2, \ldots, 16\}$ the set of items. Item i weighs $(14+i)/15$ kg for $i \in I$. The warehouse receives orders, each of which specifies four distinct items, randomly sampled from I. *Lanes* connect adjacent bins; Figure 6 illustrates the set of all possible lanes. Upon receiving an order, an unmanned vehicle solves a traveling salesman problem to calculate the shortest path to start from its base at node 0, pick four items, and finally return to node 0. We assume that the vehicle always starts by traveling from base node 0 to node 1. The vehicle has an initial speed of 2.0, but every time the vehicle picks up an item, its speed is reduced by $0.2 \times$ (item weight).

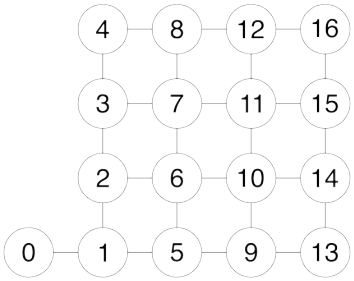


Figure 6: Warehouse layout.

The designer must consider warehouse layout design, item retrieval time, and the item placement strategy. She can choose, e.g., to build the maximal set of lanes between bins as shown in Figure 6, incurring a high construction cost, or a smaller set of lanes, which could potentially result in prohibitively high retrieval times. The items must also be carefully placed so that the vehicle minimizes the distance traveled while carrying heavy items. The objective is to jointly determine a warehouse design and item placement strategy that balances these competing concerns.

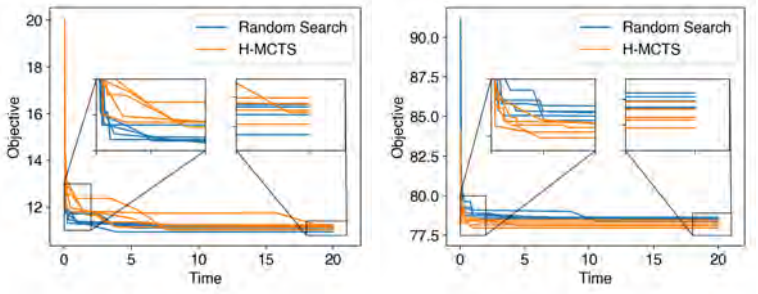
Formally, the goal is to determine the placement of items and configuration of warehouse lanes that minimizes the overall cost function $E[Y(u,v)] + \alpha \sum_{i=1}^{24} u_i$, where $u \in \{0,1\}^{24}$ is the binary vector indicating whether a lane should be built between the ith horizonally or vertically adjacent pair of bins as shown in Figure 6, v is a permutation of I indicating the placement of items, $E[Y(u,v)]$ is the mean order retrieval time depending on the item placement and the warehouse layout, and α is the cost parameter related to construction. The objective function balances the mean order retrieval time and the construction cost of the warehouse. We impose the constraint that every bin is accessible from the base station in order for a solution u to be feasible.

A GMM was trained using a set of 2,000 randomly generated warehouse designs that were then simulated to estimate the overall cost. Specifically, a feasible random graph is generated for the warehouse layout. The connections are not limited to horizontal and vertical lanes as in Figure 6; any pair of nodes (i, j) can be connected by a lane. In each bin, the weight of the item is randomly sampled from a uniform distribution on $[1.0, 2.0]$. Note that this training dataset is more general than needed for the current optimization problem; this generality allows the resulting GMM to be flexibly used across multiple optimization problems.

The standard MCTS algorithm was run over a 20 second interval. At each iteration of the MCTS algorithm, the best solution obtained from the metamodel so far was simulated over many replications to precisely estimate the actual value of the cost $C(\lambda, \bar{x})$ for the current best solution. Figure 6 shows the progress of the optimization for $\alpha = 0$ and $\alpha = 4$ (orange lines). As a comparison, a sequence of random solutions was also generated and simulated to obtain the objective values, and the best solution so far was recorded at each time period (blue lines). In each plot, the five curves of each color correspond to five repetitions each of the randomized MCTS and random-selection methods. When $\alpha = 0$, MCTS converges to same solution as random search, but more slowly. When $\alpha = 4$, however, MCTS converges faster and to a better solution than random search. The reason is that when $\alpha = 0$, the problem is "easy" in that there is no cost to building lanes, so that many different solutions of similar good quality may be available, increasing the probability that a random search would find a good solution. On the other hand, as α increases, building lanes becomes costly and the problem becomes more difficult; the MCTS approach then becomes superior to random search.

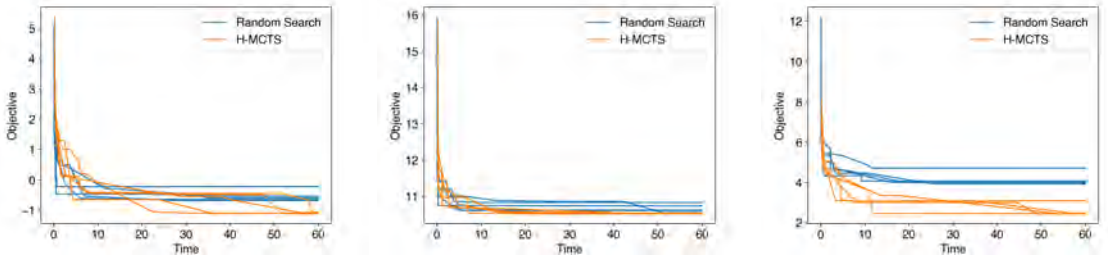
6.3 Hybrid optimization: SAN Work Rate and Layout

Our final example is the full SAN optimization problem described in Section 2, where both work rates and edge removals (use of vendors) are to be optimized. As in Section 6.1, we restrict each λ_i to lie in the range $[0.25, 1.0]$ and take take $\beta = 6$ in the cost function given in Equation (1); for the current experiment



(a) $\alpha = 0$. (b) $\alpha = 4$.

Figure 7: Simulated objective values for warehouse optimization as optimization time increases.



(a) $\alpha = 1$, two parameters. (b) $\alpha = 0.01$, two parameters. (c) $\alpha = 1$, eight parameters.

Figure 8: Simulated objective values for hybrid SAN optimization as optimization time increases.

we consider several values of α. Using the same GMM as in Section 2, we now apply the full H-MCTS algorithm to search for optimal solutions. That is, we construct a search tree where each node represents a decision on whether to include a particular edge in the original SAN or not, while ensuring that every activity in the SAN had at least one outgoing edge. At the terminal node of the search tree, we apply gradient descent on the work rates until they converge.

As in the last section, we plot the decreasing "ground truth" objective values (obtained via extensive simulation) for the solutions proposed by H-MCTS as the optimization time increases, this time up to 60 seconds. We also compare to a baseline random-search approach. The results are given in Figure 8, and are somewhat similar in nature to those for the warehouse model. Again, higher values of α correspond to harder optimization problems, this time because edge removal becomes expensive. For the two easier cases of optimizing only λ_1 and λ_2, the performance of H-MCTS and random search are comparable, with H-MCTS converging to a solution having slightly lower cost. A possible reason for this is that the more sophisticated search strategy of H-MCTS is initially counterbalanced by the fact that random search can explore many more solutions than MCTS given a fixed optimization time: at each step, random search only needs to sample a new random SAN and perform one forward pass of the trained GMM, whereas each MCTS iteration required roughly 10 gradient descent steps for the parameters to converge, each of which requires one forward pass and one backward pass of the trained GMM model. In contrast, for the "hard" problem with $\alpha = 1$ and optimization of all eight λ_i parameters, H-MCTS converges much more rapidly than random search, and to a markedly better solution. So our experiment indicates the potential of our approach to address difficult hybrid optimization problems.

7 CONCLUSION

Graphical metamodels provide a new way to capture the impact of both numerical parameters and simulation model structure on a performance measure of interest while avoiding expensive simulation runs. Given a GMM of a complex simulation model, this paper provides a unified algorithmic framework for performing joint optimization on both continuous and discrete decision variables, where the discrete variables can include the graphical structure of the simulation model. Our initial results indicate good accuracy and performance, facilitating operational and tactical decision making in complex, uncertain environments. In future work, we plan to conduct additional experiments on a variety of complex optimization problems to further validate our approach. We also plan to use active-learning techniques (Settles 2012) to reduce the number of simulation runs needed to train a GMM. Finally, we are investigating methods to explicitly incorporate both simulation and metamodeling uncertainty into our methodology.

REFERENCES

Barton, R. R. 2020. "Tutorial: Metamodeling for Simulation". In *Proceedings of the 2020 Winter Simulation Conference*, edited by K.-H. G. Bae, B. Feng, S. Kim, S. Lazarova-Molnar, Z. Zheng, T. Roeder, and R. Thiesing, 1102–1116. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Boesel, J., B. L. Nelson, and S.-H. Kim. 2003. "Using Ranking and Selection to "Clean Up" After Simulation optimization". *Operations Research* 51(5):814–825.

Cen, W., and P. J. Haas. 2022. "Enhanced Simulation Metamodeling via Graph and Generative Neural Networks". In *Proceedings of the 2022 Winter Simulation Conference*, edited by B. Feng, G. Pedrielli, Y. Peng, S. Shashaani, E. Song, C. Corlu, L. Lee, E. Chew, T. Roeder, and P. Lendermann, 2748–2759. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Danihelka, I., A. Guez, J. Schrittwieser, and D. Silver. 2022. "Policy Improvement by Planning with Gumbel". In *International Conference on Learning Representations*.

do Amaral, J. V. S., J. Montevechi, A. Barra, R. de Carvalho Miranda, and W. T. de Sousa Junior. 2022. "Metamodel-based Simulation Optimization: A Systematic Literature Review". *Simulation Modelling Practice and Theory* 114:102403.

Ford, M. T., S. G. Henderson, and D. J. Eckman. 2022. "Automatic Differentiation for Gradient Estimators in Simulation". In *Proceedings of the 2022 Winter Simulation Conference*, edited by B. Feng, G. Pedrielli, Y. Peng, S. Shashaani, E. Song, C. Corlu, L. Lee, E. Chew, T. Roeder, and P. Lendermann, 3134–3145. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Fu, M. C. 2018. "Monte Carlo Tree Search: A Tutorial". In *Proceedings of the 2018 Winter Simulation Conference*, edited by M. Rabe, A. A. Juan, N. Mustafee, A. Skoogh, S. Jain, and B. Johansson, 222–236. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Haas, P. J., and G. S. Shedler. 1991. "Stochastic Petri Nets: Modeling Power and Limit Theorems". *Probability in the Engineering and Informational Sciences.* 5(4):477–498.

Kim, S.-J., S. P. Boyd, S. Yun, D. D. Patil, and M. A. Horowitz. 2007. "A Heuristic for Optimizing Stochastic Activity Networks With Applications to Statistical Digital Circuit Sizing". *Optimization and Engineering* 8:397–430.

Kingma, Diederik P. and Ba, Jimmy 2014. "Adam: A Method for Stochastic Optimization". arXiv preprint arXiv:1412.6980v9.

Marti, G. 2019. "Permutation Invariance in Neural Networks". https://gmarti.gitlab.io/ml/2019/09/01/correl-invariance-permutations-nn.html, accessed 28th June, 2022.

Settles, B. 2012. *Active Learning*. Morgan & Claypool.

Simonyan, K., A. Vedaldi, and A. Zisserman. 2013. "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps". *arXiv preprint arXiv:1312.6034*.

Spall, J. C. 2003. *Introduction to Stochastic Search and Optimization: Estimation, Simulation, and Control*. Wiley.

Wang, L. 2005. "A Hybrid Genetic Algorithm–Neural Network Strategy for Simulation Optimization". *Applied Mathematics and Computation* 170(2):1329–1343.

AUTHOR BIOGRAPHIES

WANG CEN is a Ph.D. student at the University of Massachusetts Amherst, Manning College of Information and Computer Sciences. His email address is cenwang@umass.edu and his web page is https://cenwangumass.github.io.

PETER J. HAAS is a Professor at the University of Massachusetts Amherst, Manning College of Information and Computer Sciences. His email address is phaas@cs.umass.edu and his web page is https://www.cics.umass.edu/people/haas-peter.